SIL Optimizations

AllocBoxToStack

Hi, I'm Yusuke

@kitasuke

Why SIL?

Why SIL?

Why SIL?

→ Better idea of Swift type system 🍎

Why SIL?

→ Better idea of Swift type system 🍎

→ Optimizations magic 🔮✨

Why SIL?

→ Better idea of Swift type system 🍎

→ Optimizations magic 🔮✨

→ Just for fun 😎

SIL

Swift Intermediate Language

SIL Optimizations - AllocBoxToStack, Yusuke Kita (@kitasuke), iOSDC 2018

SIL is a language specific Intermediate Representation

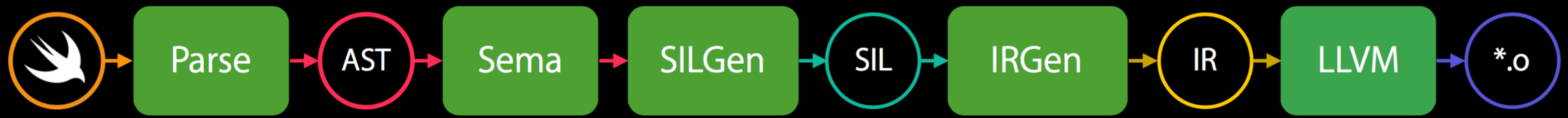
Swift

SIL Optimizations - AllocBoxToStack, Yusuke Kita (@kitasuke), iOSDC 2018

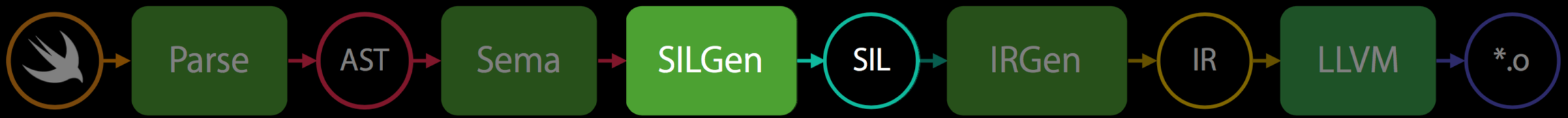
Swift Compiler

SIL Optimizations - AllocBoxToStack, Yusuke Kita (@kitasuke), iOSDC 2018

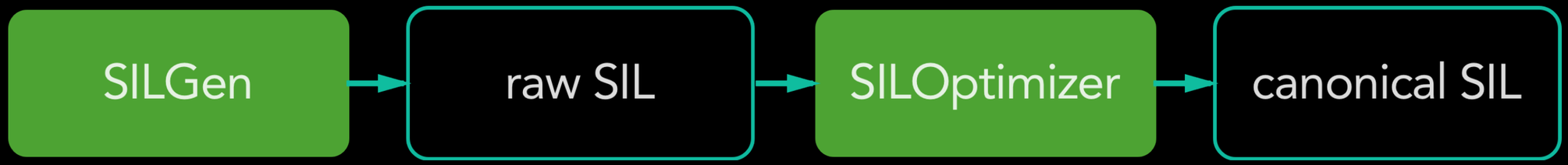
Swift Compiler



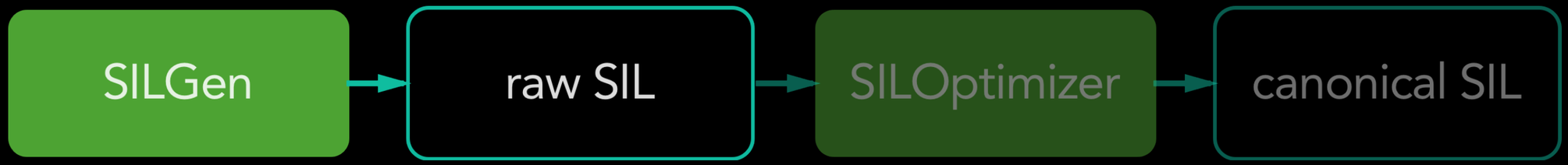
Swift Compiler



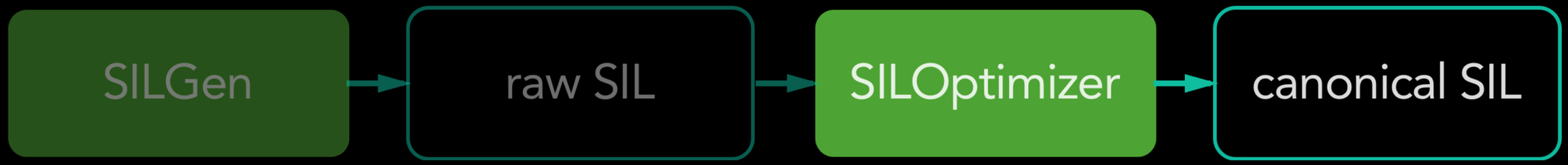
SIL



raw SIL



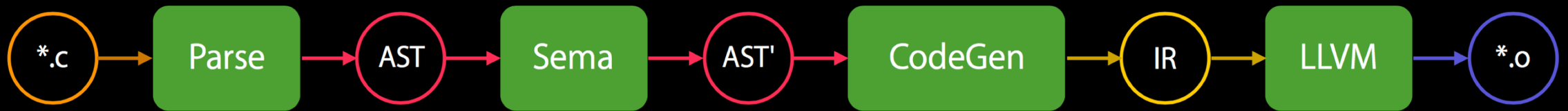
canonical SIL



Objective-C

Clang

Clang



SIL

Optimizations

SIL Optimizations - AllocBoxToStack, Yusuke Kita (@kitasuke), iOSDC 2018

Optimization Passes

→ CapturePromotion

→ AllocBoxToStack

→ MandatoryInlining

→ DeadCodeElimination

→ Mem2Reg

...

SIL Optimizations - AllocBoxToStack, Yusuke Kita (@kitasuke), iOSDC 2018

AllocBoxToStack

Stack Promotion of Box Objects

Stack vs Heap

Stack

Stack

→ Very fast

Stack

→ Very fast

→ No explicit memory management

Stack

→ Very fast

→ No explicit memory management

→ Size limits

Heap

Heap

→ Slow

Heap

→ Slow

→ Explicit memory management

Heap

→ Slow

→ Explicit memory management

→ No size limits

Stack vs Heap

Stack	Heap
Value types	Reference types
Pointers to reference type	

How AllocBoxToStack works?

alloc_box ➡ alloc_stack

stack.swift

```
func number() -> Int {
    var x: Int
    x = 1
    return x
}
```

raw SIL

```
$swiftc -emit-silgen stack.swift > stack.sil
```

stack.sil

```
%0 = alloc_box ${ var Int }, var, name "x"
%1 = mark_uninitialized [var] %0 : ${ var Int }
%2 = project_box %1 : ${ var Int }, 0
...
%10 = begin_access [read] [unknown] %2 : $*Int
%11 = load [trivial] %10 : $*Int
end_access %10 : $*Int
destroy_value %1 : ${ var Int }
return %11 : $Int
```

stack.sil

```
%0 = alloc_box ${ var Int }, var, name "x"
%1 = mark_uninitialized [var] %0 : ${ var Int }
%2 = project_box %1 : ${ var Int }, 0
...
%10 = begin_access [read] [unknown] %2 : $*Int
%11 = load [trivial] %10 : $*Int
end_access %10 : $*Int
destroy_value %1 : ${ var Int }
return %11 : $Int
```

canonical SIL

```
$swiftc -emit-sil stack.swift > stack.sil
```

stack.sil

```
%0 = alloc_stack $Int, var, name "x"
%1 = integer_literal $Builtin.Int64, 1
%2 = struct $Int (%1 : $Builtin.Int64)
...
return %2 : $Int
```

stack.sil

```
%0 = alloc_stack $Int, var, name "x"
%1 = integer_literal $Builtin.Int64, 1
%2 = struct $Int (%1 : $Builtin.Int64)
...
return %2 : $Int
```

alloc_box vs alloc_stack

alloc_box

alloc_box allocates **a reference counted box on heap**.

alloc_stack

alloc_stack allocates a value **on stack**. All allocations must **be deallocated prior to returning from a function**.

Stack Promotion of Box Objects

alloc_box ➡ ~~alloc_stack~~

alloc_box ➡ alloc_box

box.swift

```
func closure(_ completion: @escaping () -> Void) {
    completion()
}

func number() -> Int {
    var i: Int
    i = 0
    closure { i = 10 }
    return i
}
```

box.swift

```
func closure(_ completion: @escaping () -> Void) {
    completion()
}

func number() -> Int {
    var i: Int
    i = 0
    closure { i = 10 }
    return i
}
```

box.swift

```
func closure(_ completion: @escaping () -> Void) {
    completion()
}

func number() -> Int {
    var i: Int
    i = 0
    closure { i = 10 }
    return i
}
```

raw SIL

```
$swiftc -emit-silgen box.swift > box.sil
```

box.sil

```
%0 = alloc_box ${ var Int }, var, name "i"
%1 = mark_uninitialized [var] %0 : ${ var Int }
%2 = project_box %1 : ${ var Int }, 0
...
%16 = begin_access [read] [unknown] %2 : $*Int
%17 = load [trivial] %16 : $*Int
end_access %16 : $*Int
destroy_value %1 : ${ var Int }
return %17 : $Int
```

box.sil

```
%0 = alloc_box ${ var Int }, var, name "i"
%1 = mark_uninitialized [var] %0 : ${ var Int }
%2 = project_box %1 : ${ var Int }, 0
...
%16 = begin_access [read] [unknown] %2 : $*Int
%17 = load [trivial] %16 : $*Int
end_access %16 : $*Int
destroy_value %1 : ${ var Int }
return %17 : $Int
```

canonical SIL

```
$swiftc -emit-sil box.swift > box.sil
```

box.sil

```
%0 = alloc_box ${ var Int }, var, name "i"
%1 = project_box %0 : ${ var Int }, 0
%2 = integer_literal $Builtin.Int64, 0
%3 = struct $Int (%2 : $Builtin.Int64)
...
%12 = begin_access [read] [dynamic] %1 : $*Int
%13 = load %12 : $*Int
end_access %12 : $*Int
strong_release %0 : ${ var Int }
return %13 : $Int
```

box.sil

```
%0 = alloc_box ${ var Int }, var, name "i"
%1 = project_box %0 : ${ var Int }, 0
%2 = integer_literal $Builtin.Int64, 0
%3 = struct $Int (%2 : $Builtin.Int64)
...
%12 = begin_access [read] [dynamic] %1 : $*Int
%13 = load %12 : $*Int
end_access %12 : $*Int
strong_release %0 : ${ var Int }
return %13 : $Int
```

How can promote `alloc_box`?

```
class AllocBoxToStack : public SILFunctionTransform {
  void run() override {
    ...
    AllocBoxToStackState pass(this);
    for (auto &BB : *getFunction()) {
      for (auto &I : BB)
        if (auto *ABI = dyn_cast<AllocBoxInst>(&I))
          if (canPromoteAllocBox(ABI, pass.PromotedOperands))
            pass.Promotable.push_back(ABI);
    }

    if (!pass.Promotable.empty()) {
      auto Count = rewritePromotedBoxes(pass);
      ...
    }
    ...
  }
};
```

```
class AllocBoxToStack : public SILFunctionTransform {
  void run() override {
    ...
    AllocBoxToStackState pass(this);
    for (auto &BB : *getFunction()) {
      for (auto &I : BB)
        if (auto *ABI = dyn_cast<AllocBoxInst>(&I))
          if (canPromoteAllocBox(ABI, pass.PromotedOperands))
            pass.Promotable.push_back(ABI);
    }

    if (!pass.Promotable.empty()) {
      auto Count = rewritePromotedBoxes(pass);
      ...
    }
    ...
  }
};
```

```
class AllocBoxToStack : public SILFunctionTransform {
  void run() override {
    ...
    AllocBoxToStackState pass(this);
    for (auto &BB : *getFunction()) {
      for (auto &I : BB)
        if (auto *ABI = dyn_cast<AllocBoxInst>(&I))
          if (canPromoteAllocBox(ABI, pass.PromotedOperands))
            pass.Promotable.push_back(ABI);
    }

    if (!pass.Promotable.empty()) {
      auto Count = rewritePromotedBoxes(pass);
      ...
    }
    ...
  }
};
```

```
class AllocBoxToStack : public SILFunctionTransform {
  void run() override {
    ...
    AllocBoxToStackState pass(this);
    for (auto &BB : *getFunction()) {
      for (auto &I : BB)
        if (auto *ABI = dyn_cast<AllocBoxInst>(&I))
          if (canPromoteAllocBox(ABI, pass.PromotedOperands))
            pass.Promotable.push_back(ABI);
    }

    if (!pass.Promotable.empty()) {
      auto Count = rewritePromotedBoxes(pass);
      ...
    }
    ...
  }
};
```

```
static SILInstruction * findUnexpectedBoxUse(SILValue Box, bool examinePartialApply,
    bool inAppliedFunction, llvm::SmallVectorImpl<Operand *> &PromotedOperands) {

  llvm::SmallVector<Operand *, 4> LocalPromotedOperands;
  llvm::SmallVector<Operand *, 32> Worklist(Box->use_begin(), Box->use_end());
  while (!Worklist.empty()) {
    auto *Op = Worklist.pop_back_val();
    auto *User = Op->getUser();

    if (isa<StrongRetainInst>(User) || isa<StrongReleaseInst>(User) ||
        isa<ProjectBoxInst>(User) || isa<DestroyValueInst>(User) ||
        (!inAppliedFunction && isa<DeallocBoxInst>(User)))
      continue;

    if (isa<MarkUninitializedInst>(User) || isa<CopyValueInst>(User)) {
      copy(cast<SingleValueInstruction>(User)->getUses(), std::back_inserter(Worklist));
      continue;
    }

    if (auto *PAI = dyn_cast<PartialApplyInst>(User))
      if (examinePartialApply && checkPartialApplyBody(Op) &&
        !partialApplyEscapes(PAI, /* examineApply = */ true)) {
        LocalPromotedOperands.push_back(Op);
        continue;
      }
    return User;
  }
  ...
}
```

```
static SILInstruction * findUnexpectedBoxUse(SILValue Box, bool examinePartialApply,
    bool inAppliedFunction, llvm::SmallVectorImpl<Operand *> &PromotedOperands) {

  llvm::SmallVector<Operand *, 4> LocalPromotedOperands;
  llvm::SmallVector<Operand *, 32> Worklist(Box->use_begin(), Box->use_end());
  while (!Worklist.empty()) {
    auto *Op = Worklist.pop_back_val();
    auto *User = Op->getUser();

    if (isa<StrongRetainInst>(User) || isa<StrongReleaseInst>(User) ||
        isa<ProjectBoxInst>(User) || isa<DestroyValueInst>(User) ||
        (!inAppliedFunction && isa<DeallocBoxInst>(User)))
      continue;

    if (isa<MarkUninitializedInst>(User) || isa<CopyValueInst>(User)) {
      copy(cast<SingleValueInstruction>(User)->getUses(), std::back_inserter(Worklist));
      continue;
    }

    if (auto *PAI = dyn_cast<PartialApplyInst>(User))
      if (examinePartialApply && checkPartialApplyBody(Op) &&
        !partialApplyEscapes(PAI, /* examineApply = */ true)) {
        LocalPromotedOperands.push_back(Op);
        continue;
      }
    return User;
  }
  ...
}
```

```
static SILInstruction * findUnexpectedBoxUse(SILValue Box, bool examinePartialApply,
    bool inAppliedFunction, llvm::SmallVectorImpl<Operand *> &PromotedOperands) {

  llvm::SmallVector<Operand *, 4> LocalPromotedOperands;
  llvm::SmallVector<Operand *, 32> Worklist(Box->use_begin(), Box->use_end());
  while (!Worklist.empty()) {
    auto *Op = Worklist.pop_back_val();
    auto *User = Op->getUser();

    if (isa<StrongRetainInst>(User) || isa<StrongReleaseInst>(User) ||
        isa<ProjectBoxInst>(User) || isa<DestroyValueInst>(User) ||
        (!inAppliedFunction && isa<DeallocBoxInst>(User)))
      continue;

    if (isa<MarkUninitializedInst>(User) || isa<CopyValueInst>(User)) {
      copy(cast<SingleValueInstruction>(User)->getUses(), std::back_inserter(Worklist));
      continue;
    }

    if (auto *PAI = dyn_cast<PartialApplyInst>(User))
      if (examinePartialApply && checkPartialApplyBody(Op) &&
        !partialApplyEscapes(PAI, /* examineApply = */ true)) {
        LocalPromotedOperands.push_back(Op);
        continue;
      }
    return User;
  }
  ...
}
```

```
static SILInstruction * findUnexpectedBoxUse(SILValue Box, bool examinePartialApply,
    bool inAppliedFunction, llvm::SmallVectorImpl<Operand *> &PromotedOperands) {

  llvm::SmallVector<Operand *, 4> LocalPromotedOperands;
  llvm::SmallVector<Operand *, 32> Worklist(Box->use_begin(), Box->use_end());
  while (!Worklist.empty()) {
    auto *Op = Worklist.pop_back_val();
    auto *User = Op->getUser();

    if (isa<StrongRetainInst>(User) || isa<StrongReleaseInst>(User) ||
        isa<ProjectBoxInst>(User) || isa<DestroyValueInst>(User) ||
        (!inAppliedFunction && isa<DeallocBoxInst>(User)))
      continue;

    if (isa<MarkUninitializedInst>(User) || isa<CopyValueInst>(User)) {
      copy(cast<SingleValueInstruction>(User)->getUses(), std::back_inserter(Worklist));
      continue;
    }

    if (auto *PAI = dyn_cast<PartialApplyInst>(User))
      if (examinePartialApply && checkPartialApplyBody(Op) &&
        !partialApplyEscapes(PAI, /* examineApply = */ true)) {
        LocalPromotedOperands.push_back(Op);
        continue;
      }
    return User;
  }
  ...
}
```

```
static SILInstruction * findUnexpectedBoxUse(SILValue Box, bool examinePartialApply,
    bool inAppliedFunction, llvm::SmallVectorImpl<Operand *> &PromotedOperands) {

  llvm::SmallVector<Operand *, 4> LocalPromotedOperands;
  llvm::SmallVector<Operand *, 32> Worklist(Box->use_begin(), Box->use_end());
  while (!Worklist.empty()) {
    auto *Op = Worklist.pop_back_val();
    auto *User = Op->getUser();

    if (isa<StrongRetainInst>(User) || isa<StrongReleaseInst>(User) ||
        isa<ProjectBoxInst>(User) || isa<DestroyValueInst>(User) ||
        (!inAppliedFunction && isa<DeallocBoxInst>(User)))
      continue;

    if (isa<MarkUninitializedInst>(User) || isa<CopyValueInst>(User)) {
      copy(cast<SingleValueInstruction>(User)->getUses(), std::back_inserter(Worklist));
      continue;
    }

    if (auto *PAI = dyn_cast<PartialApplyInst>(User))
      if (examinePartialApply && checkPartialApplyBody(Op) &&
        !partialApplyEscapes(PAI, /* examineApply = */ true)) {
        LocalPromotedOperands.push_back(Op);
        continue;
      }
    return User;
  }
  ...
}
```

Recap

→ Promotes **@noescape** values to stack

→ **@escaping** values with a reference count

Optimization Flags

optimized.swift

```
func closure(_ completion: @escaping () -> Void) {
    completion()
}

func number() -> Int {
    var i: Int
    i = 0
    closure { i = 10 }
    return i
}
```

canonical SIL with -O

```
$swiftc -emit-sil -O optimized.swift > optimized.sil
```

optimized.sil

```
%0 = integer_literal $Builtin.Int64, 10
%1 = struct $Int (%0 : $Builtin.Int64)
return %1 : $Int
```

No *alloc_box* or *alloc_stack*

Mem2Reg

Register Promotion of Stack Allocations

How can promote stack allocation?

```
class SILMem2Reg : public SILFunctionTransform {
  void run() override {
    SILFunction *F = getFunction();

    DominanceAnalysis* DA = PM->getAnalysis<DominanceAnalysis>();
    bool Changed = MemoryToRegisters(*F, DA->get(F)).run();
    ...
  }
};
```

```
class SILMem2Reg : public SILFunctionTransform {
  void run() override {
    SILFunction *F = getFunction();

    DominanceAnalysis* DA = PM->getAnalysis<DominanceAnalysis>();
    bool Changed = MemoryToRegisters(*F, DA->get(F)).run();
    ...
  }
};
```

```
class SILMem2Reg : public SILFunctionTransform {
  void run() override {
    SILFunction *F = getFunction();

    DominanceAnalysis* DA = PM->getAnalysis<DominanceAnalysis>();
    bool Changed = MemoryToRegisters(*F, DA->get(F)).run();
    ...
  }
};
```

```
class SILMem2Reg : public SILFunctionTransform {
  void run() override {
    SILFunction *F = getFunction();

    DominanceAnalysis* DA = PM->getAnalysis<DominanceAnalysis>();
    bool Changed = MemoryToRegisters(*F, DA->get(F)).run();
    ...
  }
};
```

```
bool MemoryToRegisters::promoteSingleAllocation(AllocStackInst *alloc, DomTreeLevelMap &DomTreeLevels){
  NumAllocStackFound++;

  bool inSingleBlock = false;
  if (isCaptured(alloc, inSingleBlock)) {
    NumAllocStackCaptured++;
    return false;
  }

  if (isWriteOnlyAllocation(alloc)) {
    eraseUsesOfInstruction(alloc);
    return true;
  }

  if (inSingleBlock) {
    removeSingleBlockAllocation(alloc);

    if (!alloc->use_empty()) {
      B.setInsertionPoint(std::next(alloc->getIterator()));
      B.createDeallocStack(alloc->getLoc(), alloc);
    }
    return true;
  }

  StackAllocationPromoter(alloc, DT, DomTreeLevels, B).run();
  eraseUsesOfInstruction(alloc);
  return true;
}
```

```
bool MemoryToRegisters::promoteSingleAllocation(AllocStackInst *alloc, DomTreeLevelMap &DomTreeLevels){
  NumAllocStackFound++;

  bool inSingleBlock = false;
  if (isCaptured(alloc, inSingleBlock)) {
    NumAllocStackCaptured++;
    return false;
  }

  if (isWriteOnlyAllocation(alloc)) {
    eraseUsesOfInstruction(alloc);
    return true;
  }

  if (inSingleBlock) {
    removeSingleBlockAllocation(alloc);

    if (!alloc->use_empty()) {
      B.setInsertionPoint(std::next(alloc->getIterator()));
      B.createDeallocStack(alloc->getLoc(), alloc);
    }
    return true;
  }

  StackAllocationPromoter(alloc, DT, DomTreeLevels, B).run();
  eraseUsesOfInstruction(alloc);
  return true;
}
```

```
bool MemoryToRegisters::promoteSingleAllocation(AllocStackInst *alloc, DomTreeLevelMap &DomTreeLevels){
  NumAllocStackFound++;

  bool inSingleBlock = false;
  if (isCaptured(alloc, inSingleBlock)) {
    NumAllocStackCaptured++;
    return false;
  }

  if (isWriteOnlyAllocation(alloc)) {
    eraseUsesOfInstruction(alloc);
    return true;
  }

  if (inSingleBlock) {
    removeSingleBlockAllocation(alloc);

    if (!alloc->use_empty()) {
      B.setInsertionPoint(std::next(alloc->getIterator()));
      B.createDeallocStack(alloc->getLoc(), alloc);
    }
    return true;
  }

  StackAllocationPromoter(alloc, DT, DomTreeLevels, B).run();
  eraseUsesOfInstruction(alloc);
  return true;
}
```

```
bool MemoryToRegisters::promoteSingleAllocation(AllocStackInst *alloc, DomTreeLevelMap &DomTreeLevels){
  NumAllocStackFound++;

  bool inSingleBlock = false;
  if (isCaptured(alloc, inSingleBlock)) {
    NumAllocStackCaptured++;
    return false;
  }

  if (isWriteOnlyAllocation(alloc)) {
    eraseUsesOfInstruction(alloc);
    return true;
  }

  if (inSingleBlock) {
    removeSingleBlockAllocation(alloc);

    if (!alloc->use_empty()) {
      B.setInsertionPoint(std::next(alloc->getIterator()));
      B.createDeallocStack(alloc->getLoc(), alloc);
    }
    return true;
  }

  StackAllocationPromoter(alloc, DT, DomTreeLevels, B).run();
  eraseUsesOfInstruction(alloc);
  return true;
}
```

```
bool MemoryToRegisters::promoteSingleAllocation(AllocStackInst *alloc, DomTreeLevelMap &DomTreeLevels){
  NumAllocStackFound++;

  bool inSingleBlock = false;
  if (isCaptured(alloc, inSingleBlock)) {
    NumAllocStackCaptured++;
    return false;
  }

  if (isWriteOnlyAllocation(alloc)) {
    eraseUsesOfInstruction(alloc);
    return true;
  }

  if (inSingleBlock) {
    removeSingleBlockAllocation(alloc);

    if (!alloc->use_empty()) {
      B.setInsertionPoint(std::next(alloc->getIterator()));
      B.createDeallocStack(alloc->getLoc(), alloc);
    }
    return true;
  }

  StackAllocationPromoter(alloc, DT, DomTreeLevels, B).run();
  eraseUsesOfInstruction(alloc);
  return true;
}
```

optimized.swift

```
func closure(_ completion: @escaping () -> Void) {
    completion()
}

func number() -> Int {
    var i: Int
    i = 0
    closure { i = 10 }
    return i
}
```

optimized.swift

```
func number() -> Int {
    return 10
}
```

Optimization Flags

→ `-Onone`

→ `-O`

→ `-Ounchecked`

→ `-Osize`

Tips for Debug

→ `-Xllvm -sil-print-all`

→ `-Xllvm -sil-print-only-functions`

→ `-Xllvm -sil-print-before/after/around`

Summary

→ Optimize, optimize and optimize 💪

→ Better ideas of how Swift Compiler works 💯

→ Definitely worth learning 🏆

References

→ var vs let in SIL

→ swift/docs/SIL.rst

→ Debugging the Swift Compiler

→ Swift's High-Level IR: A Case Study of Complementing LLVM IR with Language-Specific Optimization

Thank you!